DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	bianchi-mariano@hotmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El primer problema de este trabajo práctico consistió en dar un algoritmo que sea capaz de decir, dada una secuencia de números, la mínima cantidad de números a eliminar de la misma para que ésta sea unimodal. Esto es, que la secuencia cumple que es creciente desde el primer elemento hasta cierta posición y desde dicha posición es decreciente hasta el final[1].

Además, se estableció como requerimiento que la complejidad del algoritmo en cuestión fuera estrictamente menor que $O(n^3)$, siendo n el tamaño de la secuencia.

Se pensaron varias formas de encarar y resolver este problema, finalmente se aplicó la técnica de *programación dinámica*. En las sucesivas secciones, daremos una explicación detallada de cómo se implementó el algoritmo, cuál es su complejidad temporal y cómo se comportó frente a distintos valores y tamaños de entrada.

1.2. Explicación

Desde un comienzo se pensó que el problema podía resolverse usando *programación dinámica*, sin embargo comprender de qué forma debía aplicarse *principio de optimalidad* no fue, en principio, nada trivial. Tras un tiempo de analizar el problema, se pudo vislumbrar que el mismo podía ser visto como una variación de otro problema similar: el de encontrar dentro de una secuencia la subsecuencia creciente/decreciente más larga.
Básicamente, el razonamiento empleado fue el siguiente:

Si para cada posición i de la secuencia se conoce:

- Longitud de la subsecuencia creciente más larga desde el principio hasta i (que contenga al elemento i-ésimo)
- Longitud de la subsecuencia decreciente más larga desde i hasta el final (que contenga al elemento i-ésimo)

entonces es posible decir cuál es el largo de una secuencia unimodal que tiene como "pivote" u objeto más grande al i-ésimo de la secuencia. Finalmente, sólo bastaría averiguar cuál es el "pivote" que maximice esa longitud para de ésta forma resolver el problema.
Cabe observar, que determinar la longitud de la subsecuencia unimodal más larga es equivalente a encontrar la cantidad mínima de elementos a eliminar para transformar la secuencia dada en una secuencia unimodal. Este dado será tenido en cuenta a la hora de implementar el algoritmo.

A continuación se desarrolla el funcionamiento del algoritmo mediante el cual se logra resolver el problema planteado.

Para construir el algoritmo de programación dinámica se definieron:

[1] Debe ser estrictamente creciente/decreciente

- un vector denominado *ascenso* donde se va guardando, en la i-ésima posicion, la longitud de la subsecuencia más larga desde el inicio de la secuencia original hasta la posición i. Esto es:
  $ascenso_i$ = longitud de la secuencia creciente más larga que termina con el número v_i
- la siguiente relación de recurrencia:
  - $ascenso_0 = 0$
  - $ascenso_i = max_{j<i}\{ascenso_j + 1\}$ (con $v_j < v_i$)
- y la solucion final: $max_{1 \leq i \leq n}\{ascenso_i\}$

Para entender el algoritmo principal, se presenta el pseudocódigo de la función *completar_ascensos*, la cual completa el vector *ascenso* con los valores correspondientes.

```
1 void completar_ascensos(ascenso: vector< nat >, v: vector< int >)
2 Complejidad: θ(tam(v)^2)
3 for (i = 0; i <= tam(v); i++);                          // θ(tam(v)^2))   a
4 do
5 |  ascenso[i] ← maximoAsc(ascenso,v,i) + 1;                              // θ(i)
6 end
```

[a] esta complejidad se da ya que $\sum_{i=0}^{tam(v)} (i) = \frac{tam(v) * (tam(v) - 1)}{2} \in \theta(tam(v)^2)$

La función maximoAsc(ascenso,v,i) indica, dentro del vector ascenso, la maxima longitud de un ascenso hasta la posicion i-1 del vector tal que cumpla la siguiente propiedad:

Si j es el indice del ascenso donde se encuentra el resultado, tiene que ocurrir que: [2]

$$(\forall k \in [0...i), v[k] < v[i])\ ascenso[j] > ascenso[k]$$

Por lo tanto, de esta manera podemos obtener para cada pivote, la longitud del mayor ascenso hasta esa posición.

Veamos un *ejemplo*:
$v = \{9, 5, 2, 8, 7, 3, 1, 6, 4\}$
Las subsecuencias ascendentes más largas son $\{2, 3, 4\}$ o $\{2, 3, 6\}$

sucesión	9	5	2	8	7	3	1	6	4
longitud	1	1	1	2	2	2	1	3	3

Aquí podemos observar que en cada posición de la fila longitud, se encuentra la longitud de la secuencia creciente más larga hasta ese punto.

[2] ver pseudocódigo en el anexo

Una vez realizado este proceso, podemos obtener, ejecutando un algoritmo equivalente (pero desde el final hacia el principio) que obtenga, para cada posición, la longitud de la secuencia decreciente más larga desde la posición hacia el final. Al ejecutar esté proceso sobre el ejemplo obtendríamos:

sucesión	9	5	2	8	7	3	1	6	4
ascendente	1	1	1	2	2	2	1	3	3
descendente	5	3	2	4	3	2	1	2	1

Luego, para decidir cual es el mejor pivot, nos basamos en dos filas extra implicitas que podemos ver a continuación :

sucesión	9	5	2	8	7	3	1	6	4
ascendente	1	1	1	2	2	2	1	3	3
descendente	5	3	2	4	3	2	1	2	1
suma	5	3	2	5	4	3	1	4	3
tam(S) - suma;	4	6	7	4	5	6	8	5	6

Donde $suma_i = ascendente_i + descendente_i - 1 \ \forall i \in [1..tam(s)]$
(esto quiere decir, la suma de las longitudes de la secuencia de mayor longitud creciente hasta i + la longitud de la secuencia de mayor longitud decreciente desde i). Se le resta 1 porque ambas secuencias incluyen al elemento de la posición i.

La fila tam(S)-suma representa, cuantos elementos NO fueron usados para esta "escalera".

Por lo tanto se busca el máximo valor de la fila suma, lo que es equivalente a buscar el mínimo en la fila tam(s)-suma que representa la cantidad minima de borrados a realizar para lograr formar la secuencia unimodal y ese es el resultado devuelto por nuestra función.

1.3. Análisis de la complejidad del algoritmo

A continuación se calculará la complejidad del algortimo implementado en la función *escalerar*, que es el que resuelve el problema que se planteo. Primero veamos el pseudocódigo de la función completa:

void **escalerar**(v: vector⟨int⟩)

```
1  Complejidad: θ(tam(v)^2)
2  var max: nat;
3  var ascenso: vector⟨nat⟩(tam(v));                                   // θ(tam(v))
4  var descenso: vector⟨nat⟩(tam(v));                                  // θ(tam(v))
5  completar_ascensos(ascenso,v);                                      // θ(tam(v)^2)
6  completar_descensos(descenso,v);                                    // θ(tam(v)^2)
7  for (i = 0; i ≤ tam(v); i++) ;                                      // θ(tam(v))
8  do
9  |  if (ascenso[i] + descenso[i] ≥ max) then max ← ascenso[i] + descenso[i];
   |  // θ(1)
10 end
11 retornar tam(v) − max + 1;                                          // θ(1)
```

Pseudocódigo de la función completar_descensos:

void **completar_descensos**(descenso: vector⟨nat⟩, v: vector⟨int⟩)

```
1  Complejidad: θ(tam(v)^2)
2  for (i = tam(v); i ≥ 0; i−−) ;                                      // θ(tam(v)^2)   [a]
3  do
4  |  descenso[i] ← maximoDes(descenso,v,i) + 1;                       // θ(tam(v) − i)
5  end
```

[a] esta complejidad se da ya que $\sum_{i=tam(v)}^{0} (tam(v) - i) = \sum_{i=0}^{tam(v)} (i) = \frac{tam(v) * (tam(v) - 1)}{2} \in \theta(tam(v)^2)$

Pseudocódigo de la función maximoAsc:

nat **maximoAsc**(ascenso: vector⟨nat⟩, v: vector⟨int⟩,i: nat)

```
1  Complejidad: θ(i)
2  var res: nat ;                                                      // θ(1)
3  res ← 0 ;                                                           // θ(1)
4  for (k = 0; k < i; k++);                                            // θ(i)
5  do
6  |  if (v[j] < v[i] and res < ascenso[j]) then res←ascenso[j];       // θ(1)
7  end
```

Aclaración: el pseudocódigo de la función competar_ascensos() se encuentra en la sección "Explicación" y la función maximoDes() es analoga a maximoAsc().

Como se ve en los pseudocódigos, sin depender de los valores ingresados en el vector v, la cantidad de operaciones es la misma para un mismo tamaño de entrada. Por lo tanto, podemos decir que esta función esta acotada por arriba y por abajo por una función cuadratica a partir de un n_0 es decir, escalerar() $\in \Theta(tam(v)^2)$

En conclusión, nuestro algoritmo esta siempre acotado por arriba y por abajo por una función cuadratica en el tamaño de la entrada (tam(v)).

1.4. Detalles de implementación

Para compilar el programa sólo hace falta ejecutar el comando make en consola.

Modo de Uso

1.5. Resultados

Para poder analizar el comportamiento del algoritmo, se desarrolló un generador de secuencias de tamaños determinados con valores enteros en un rango determinado.

Al realizar el análisis sobre la complejidad del algoritmo, notamos que sin importar los valores en la secuencia ni la cantidad de elementos a sacar para convertirla en unimodal, debería comportarse de manera identica para tamaños de secuencia similares.

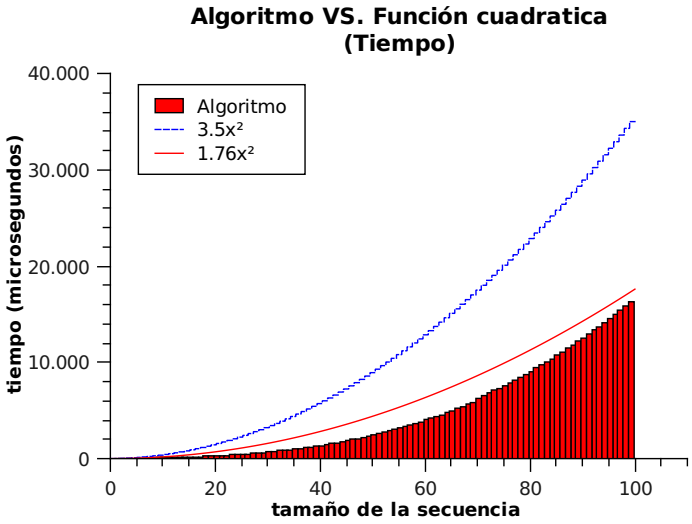
Por lo tanto, veamos que entradas generamos:

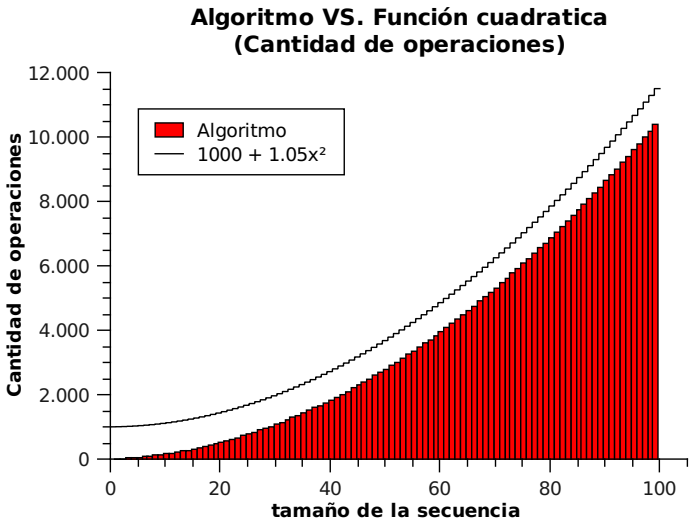
1. 100 secuencias de tamaños desde 1 hasta 99 con elementos aleatorios entre -1000000 y 1000000.
2. 250 secuencias de tamaños desde 1 hasta 250 con elementos aleatorios entre -1000000 y 1000000.
3. 250 secuencias de tamaños desde 1 hasta 250 con elementos iguales a cero, de manera que el resultado sea eliminar todos menos un elemento para convertirla en unimodal.

Luego, se desprenden algunas hipótesis, las cuales enunciaremos a continuación:

- La cantidad de operaciones no debería ser afectada por la cantidad de elementos a borrar para convertir la secuencia.
- Para tamaños de entradas identicos, el algoritmo debería realizar exactamente la misma cantidad de operaciones.

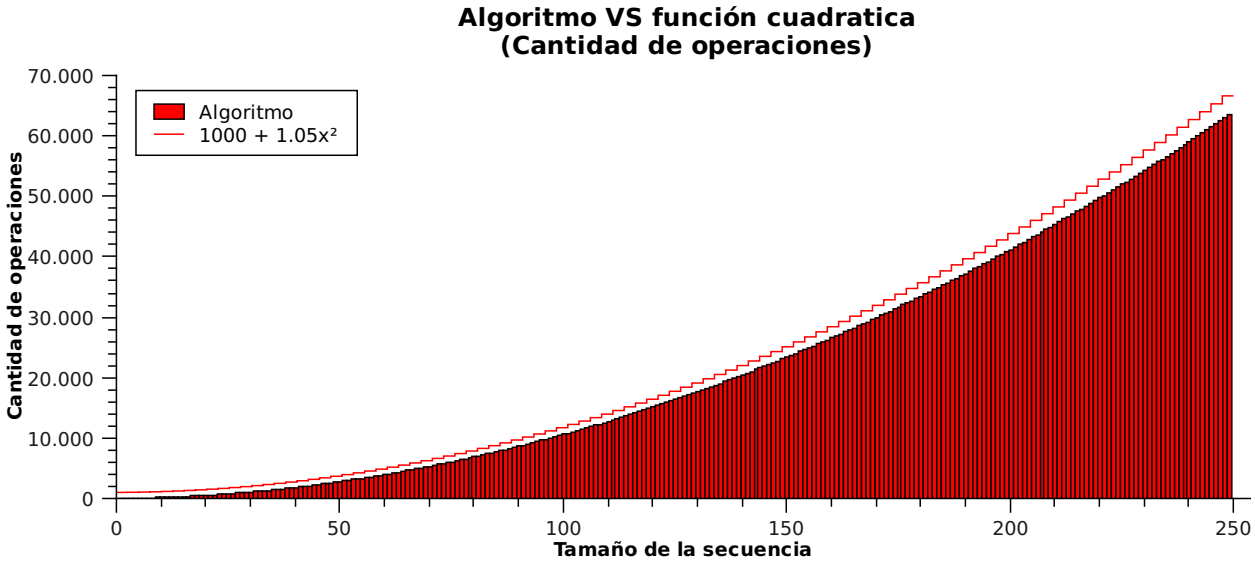
A continuación veremos gráficos que muestran el comportamiento del algoritmo utilizado



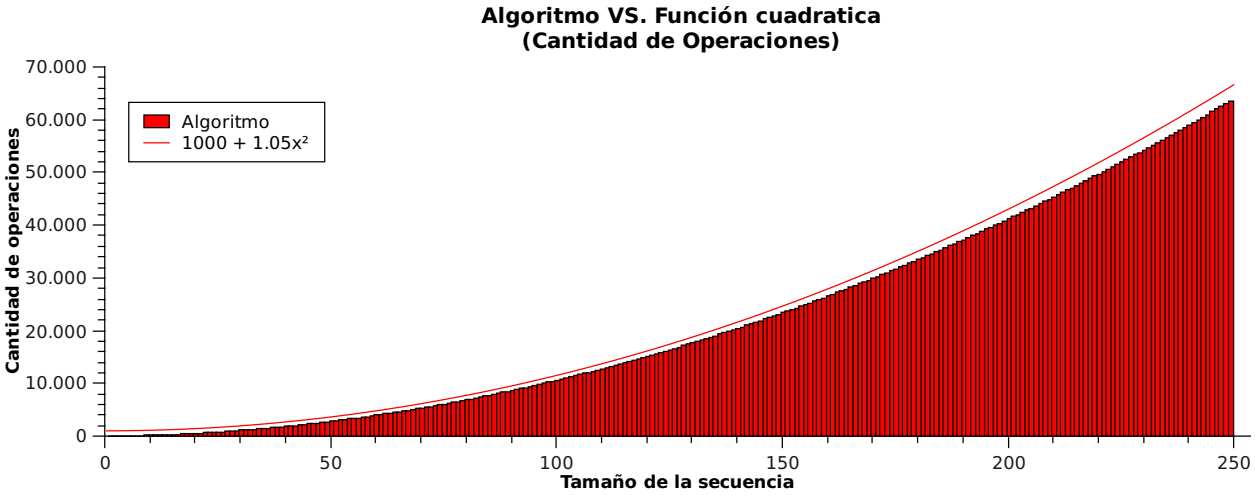


Estos gráficos muestran el tiempo de ejecución y cantidad de operaciones en secuencias de 1 hasta 100 elementos aleatorios comparados con funciones cuadraticas (se corresponden

con la entrada generada número “1” de las enunciadas anteriormente).

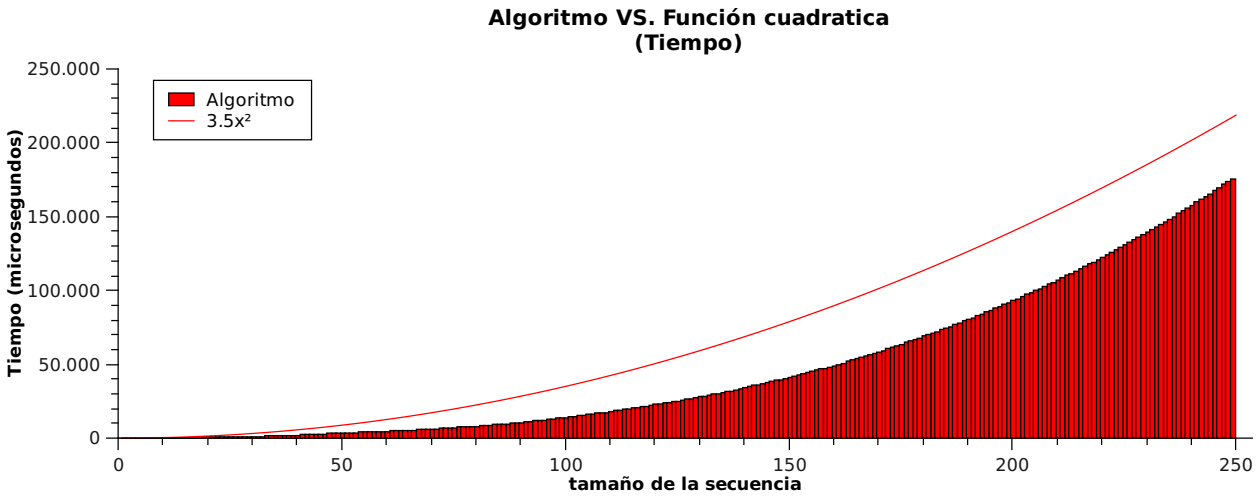


Este gráfico muestra la cantidad de operaciones en secuencias de 1 hasta 250 elementos aleatorios comparados con funciones cuadraticas (se corresponde con la entrada generada número “2” de las enunciadas anteriormente).



Este gráfico muestra la cantidad de operaciones en secuencias de 1 hasta 250 elementos iguales a cero comparados con funciones cuadraticas (se corresponde con la entrada generada

número “3” de las enunciadas anteriormente).



Este gráfico muestra la cantidad de operaciones en secuencias de 1 hasta 250 elementos comparados con funciones cuadraticas (se corresponde con la entrada generada número “3” de las enunciadas anteriormente).

Notar que para el mismo tipo de gráficos (de tiempo o de cantidad de operaciones), se utiliza las mismas funciónes cuadraticas para gráficar la cota superior e inferior que en el resto más alla de ser diferentes casos de pruebas.

1.6. Debate

Se puede apreciar en los gráficos propuestos en la sección [1.5] que en general la implementación realizada se comporta como se esperaba que en teoría lo hiciera. Es decir, durante el desarrollo del informe sobre este ejercicio, entre otras cosas, se postuló la hipótesis de que la complejidad del algoritmo tenía un costo de $\theta(tam(v)^2)$, donde $tam(v)$ es la cantidad de números en la secuencia.

Se puede identificar en cada gráfico una marcada similitud entre los resultados arrojados en las distintas mediciones que se hicieron sobre la implementación unas parabola (o funciónes cuadráticas) cuyas pendientes no varian para cada caso (medidos de la misma manera).

1.7. Conclusiones

Luego de describir el funcionamiento del algoritmo, de realizar las pruebas y gráficos pertinentes y de analizarlos detalladamente, podemos realizar algunas conclusiones.
Podemos asegurar fehacientemente que la complejidad del algoritmo propuesto es cuadratico y que la misma es $\Theta(n)$ donde n es la cantidad de trabajadores. Esto no sólo se desprende del análisis teórico realizado anteriormente, sino también de las sucesivas pruebas realizadas.

Claramente se puede observar en todos los gráficos presentados cómo el comportamiento de la implementación se asemeja a una función lineal sobre los datos de entrada.

Finalmente, si hacemos una comparación entre los resultados obtenidos al aplicarse el algoritmo sobre datos de entrada azarosos y datos de entrada que se condicen con el peor caso, se puede observar que la constante que acompaña a la complejidad para el peor caso es considerablemente mayor a la constante que aparece en el caso azaroso. Como una conclusión poco menos significativa entonces, se puede decir que los casos en que los horarios de los trabajadores son todos disjuntos son los peores casos para este algoritmo.

2. Ejercicio 2

2.1. Introducción

2.2. Explicación

Primero, algunas definiciones:

Def$_1$:
Un grafo G es fuertemente orientable si existe una asignación de direcciones a los ejes del conjunto de ejes del grafo G tal que el digrafo resultante es fuertemente conexo.

Def$_2$:
un puente, **arista de corte** o istmo es una arista que al eliminarse de un grafo incrementa el número de componentes conexos de éste. Equivalentemente, una arista es un puente si y sólo si no está contenido en ningún ciclo.

Teorema [Robbins, 1939] :
Un grafo conexo G es fuertemente orientable si y solo si G no tiene puentes (Demostración en el anexo).

Con esté teorema podemos ver que si encontramos al menos 1 puente en nuestro grafo, significa que no podremos orientarlo como queremos, y de lo contrario, si encontramos que no hay ningun puente, podremos orientarlo. Por lo tanto, el algoritmo utilizado realiza exactamente esa comprobación. Veamos como trabaja:

comprobación(Grafo G)

```
1 Complejidad: O(n^3)
2 var eje : int ← 0
3 var n : int ← cantNodos(G)
4 while eje ≤ m do
5 |   k ← RecorridoSinEje(eje,G)
6 |   if k ≠ n then return no se puede
7 |   eje ← eje + 1
8 end
9 return fuertemente conexo
```

RecorridoSinEje(eje,G) es una funcion que recorre el grafo G (con BFS o DFS) sin utilizar la arista eje y retorna la cantidad de nodos visitados, k. Como la forma de recorrer utilizada, solo recorre nodos conectados a la raiz (es decir, al nodo donde comienza el recorrido), quiere decir que el resultado k va a ser n (la cantidad total de nodos de G) si G es conexo. De lo

contrario, si k es menor que n, estamos en presencia de 2 componentes conexas (o más en el caso de $eje = 0$). Por lo tanto, el eje sacado, era un puente.

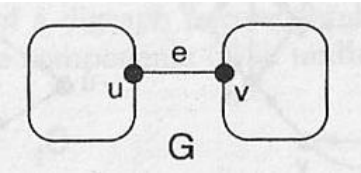
Aclaración:
Cuando eje = 0, representa, recorrer a G completo con todos sus nodos. En este punto podria pasar que G no sea conexo y esta función devolvería el resultado correspondiente.

2.3. Anexo: Demostraciones

Teorema [Robbins, 1939]:
Un grafo conexo G es fuertemente orientable si y solo si G no tiene puentes.

Demostración: $\rightarrow$)

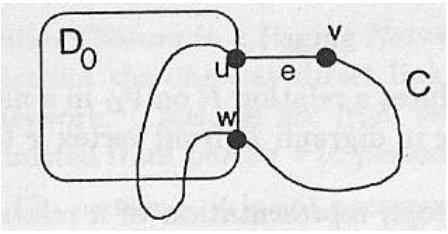
Utilizando el contrareciproco, supongamos que el grafo G tiene una arista de corte e que une a los vertices u y v. Entonces el unico camino entre u y v o v y u en el grafo G es e (ver figura). Por lo tanto para cualquier asignacion de direcciones, el nodo cola(e) nunca va a poder ser alcanzada por el nodo cabeza(e) (ver cuadro 1).



Cuadro 1:

Demostración: $\leftarrow$)

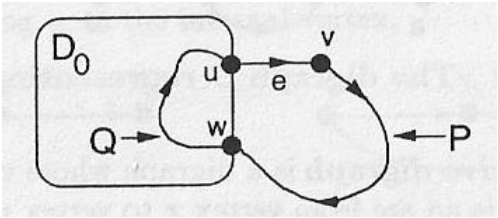
Supongamos que G es un nodo conexo sin aristas de corte. Por esto toda arista en G cae en un circuito de G. Para hacer que G sea fuertemente orientable, empezaremos con cualquier circuito (D_0) de G y dirigiremos sus ejes en una dirección (obteniendo un circuito dirigido). Si el ciclo D_0 contiene todos los nodos de G, entonces la orientación esta completa (ya que D_0 es fuertemente conexo). De lo contrario, hay que elegir cualquier eje e uniendo a un vertice u en D_0 y a un vertice v en $V_G - V_{D_0}$ (ese eje existe ya que G es conexo). Sea C = $\langle u, e, v, ..., u \rangle$ un ciclo que contiene al eje e, y sea w el primer vertice luego de v en C que cae en el ciclo D_0 (ver cuadro 2).



Cuadro 2:

A continuación, direccionamos los ejes de este camino entre v a w, obteniendo el camino dirigido $v-w$ que llamaremos P. Luego direccionamos el eje e desde u hasta v y consideramos el digrafo D_1 que se forma agregando el eje dirigido e a D_0 y todos los vertices y ejes dirigidos del camino P. Como D_0 es fuertemente conexo, entonces hay un camino dirigido de w a u (Q) en D_0 (ver cuadro 3). La concatenación de P con Q y el eje dirigido e forman un circuito simple direccionado que contiene u y los nuevos vertices de D_1. (si los vertices u y w son el mismo, entonces P satiface ser un circuito simple dirigido)

Por lo tanto, el vertice u y todos estos nuevos vetices son mutuamente alcanzables en D_1 y además u y cada vectice del digrafo D_0 tambien son mutuamente alcanzables, y, por lo tanto , el digrafo D_1 es fuertemente conexo. Este proceso puede continuar hasta que el digrafo D_1 para algun $l \geq 1$, contenga todos los vertices de G. En este punto, cualquier asignacion de direcciones hacia las aristas sin direccion restantes completaran la orientación de G, puesto que contendrá el digrafo fuertemente conexo D_l como subdigrafo.



Cuadro 3:

3. Ejercicio 3

3.1. Introducción

El tercer y último ejercicio de este trabajo práctico, consistía en dado un modelo de una prisión decidir si una persona condenada podía escapar o no de la misma. Este modelo fue dado de tal manera que se pudiera representar utilizando un grafo. La idea general del modelo de esta prisión, fue que en la misma haya habitaciones y pasillos. Cada pasillo conectaba dos habitaciones y no había más de un pasillo conectando dos habitaciones. Además, cada habitación podía estar vacía, tener una puerta o tener una llave. Para pasar por una habitación que tuviera una puerta había que tener su respectiva llave.

Como requerimiento para este problema, se especificó que su solución debía tener una complejidad estrictamente menor que $O(n^3)$, con n igual a la cantidad de habitaciones.

3.2. Explicación

En primer instancia, se pensó en alguna forma de recorrer el grafo de manera que la misma sirviera para solucionar el problema planteado. Inmediatamente, surgió la idea de utilizar el método BFS para dicho fin.

Seguidamente, se realizaron algunos ejemplos en papel, para ver de que forma se iba a comportar el método elegido para este modelo en cuestión, ya que se no podía utilizar un BFS normal por las restricciones con las que se contaba. Luego de observar este comportamiento, se encontró una forma de resolver el problema utilizando BFS, la cuál se explica a continuación.

La idea utilizada para este ejercicio resultó bastante simple. La misma consiste en los siguientes pasos:

- Se comienza a realizar BFS desde la primer habitación.
- Cada vez que se encuentra una habitación con una llave, ésta se guarda en un conjunto de llaves.
- Cada vez que se encuentra una habitación con puerta pueden suceder 2 cosas:
  - Si ya se había encontrado su llave, se la recorre como si no tuviese puerta.
  - Si no se había encontrado su llave, se la ingresa a una cola donde se alojaran las habitaciones con estas características, sin aplicar BFS en ella.
- Cuando el recorrido BFS termina, puede darse por dos motivos:
  - Que se hayan recorrido todas las habitaciones.
  - Que queden habitaciones sin recorrer en la cola de aquellas que tienen puerta y no se encontró su llave.

Una vez terminado el primer BFS, se verifica si ya se recorrieron todas las habitaciones o, caso contrario, si quedan habitaciones que se puedan visitar[3]. Si sucede lo segundo, entonces se repiten todos los pasos antes mencionados tomando como primer habitación para recorrer con BFS alguna de las que se habían encolado por no tener la llave necesaria para abrirla. Si sucede que se recorrieron todas las habitaciones o que no hay ninguna llave disponible que abra las habitaciones encoladas, entonces se sale de la recursión y se verifica si se visitó o no la última habitación (o habitación de salida). Si se visitó entonces significa que hay forma de salir de la prisión y sino, significa lo contrario.

A continuación se adjunta, a modo de pseudocódigo, los algoritmos utilizados para resolver el problema descrito anteriormente:

bool resolver(const carcel& c)

[3]esto es equivalente a ver si de todas las habitaciones que quedaron encoladas por tener una puerta a la cuál no se tenía acceso, ahora hay alguna de las que sí se tenga llave ahora

```
1  Complejidad: O(n^2)
2  vector < bool > habitacionesYaVisitadas(c.cantHabitaciones);        // O(n)
3  queue < int > habitacionesLimites;
4  vector < bool > llavesEncontradas(c.cantHabitaciones);              // O(n)
5  int proximaHabitacion;
6  bool puedoSeguir = true;                                            // O(1)
7  int contador;
8  bool termine = false;                                               // O(1)
9  proximaHabitacion = 0;                                              // O(1)
10 while (puedoSeguir ∧ ¬termine) do
11 |  recorrerPorBFS(proximaHabitacion, habitacionesLimites,
   |  habitacionesYaVisitadas, llavesEncontradas, c);                  // O(n^2)
12 |  termine = habitacionesLimites.empty();                           // O(1)
13 |  if (¬termine);                                                   // O(1)
14 |  then
15 |  |  contador = habitacionesLimites.size();                        // O(1)
16 |  |  while (puedoSeguir &&
   |  |  (¬llavesEncontradas[habitacionesLimites.front()]));          // O(n)
17 |  |  do
18 |  |  |  habitacionesLimites.push(habitacionesLimites.front());    // O(1)
19 |  |  |  habitacionesLimites.pop();                                // O(1)
20 |  |  |  contador−;                                                // O(1)
21 |  |  |  if (contador == 0);                                       // O(1)
22 |  |  |  then
23 |  |  |  |  puedoSeguir = false;                                   // O(1)
24 |  |  |  end
25 |  |  end
26 |  |  proximaHabitacion = habitacionesLimites.front();            // O(1)
27 |  |  habitacionesLimites.pop();                                  // O(1)
28 |  end
29 end
30 if (habitacionesYaVisitadas[c.cantHabitaciones-1]);                // O(1)
31 then
32 |  return true
33 else
34 |  return false
35 end
```

void recorrerPorBFS(int proximaHabitacion, *queue* < *int* > & habitacionesLimites, vector< *bool* > & habitacionesYaVisitadas, vector< *bool* > & llavesEncontradas, carcel& c)

```
1  Complejidad: O(n^2)
2  queue < int > habitacionesProximas
3  habitacionesProximas.push(proximaHabitacion)
4  habitacionesYaVisitadas[proximaHabitacion] = true;                    // O(1)
5  int actual
6  while (!habitacionesProximas.empty());                                // O(n)
7  do
8      actual = habitacionesProximas.front();                            // O(1)
9      habitacionesProximas.pop();                                       // O(1)
10     if (c.tieneLlave(actual));                                        // O(1)
11     then
12         llavesEncontradas[c.dameLlave(actual)] = true;                // O(1)
13     end
14     for (int i = 0; i<c.cantHabitaciones; i++);                       // O(n)
15     do
16         if (c.sonAdyacentes(actual,i) ∧ ¬habitacionesYaVisitadas[i]); // O(1)
17         then
18             if (c.tienePuerta(i) ∧ ¬llavesEncontradas[i]);            // O(1)
19             then
20                 habitacionesLimites.push(i);                          // O(1)
21             else
22                 habitacionesProximas.push(i);                         // O(1)
23             end
24             habitacionesYaVisitadas[i] = true;                        // O(1)
25         end
26     end
27 end
```

3.3. Análisis de la complejidad del algoritmo

Para el análisis de complejidad de los algoritmos que resuelven el problema dado, vamos a remitirnos a los pseudocódigos propuestos en la sección [3.2]. Como aclaración, cabe decir que en los siguientes párrafos vamos a referirnos a n como la cantidad de habitaciones en la prisión (cantidad de nodos del grafo).
En primer instancia vamos a analizar el pseudocódigo del algoritmo *recorrerPorBFS*. La complejidad del mismo está dada por un flujo *for* anidado dentro de otro flujo *while*. Es claramente visible que el flujo *for* tiene una complejidad $\Theta(n)$ ya que todas las operaciones dentro suyo son de complejidad $O(1)$ y el mismo se ejecuta n veces. Quizás no sea tan claro por qué el flujo *while* tiene complejidad igual a $O(n)$. Esto se debe a la naturaleza del algoritmo BFS. En BFS se recorre cada nodo una única vez, y si vemos cuándo es que termina de ejecutarse dicho flujo, veremos que lo hace cuando se vacía la cola *proximasHabitaciones*. Esta cola puede tener un largo de hasta n habitaciones (por lo dicho previamente sobre el método BFS) por lo que este flujo se ejecutará como máximo tantas veces como habitaciones haya.

Una vez realizado el análisis del algoritmo *recorrerPorBFS* sólo nos queda verificar qué complejidad cumple el algoritmo *resolver*. Si se observa con atención, puede notarse que la complejidad del mismo está dada por el llamado a la función *recorrerPorBFS*. Pero no sólo basta con decir esto. También hay un flujo *while* más adelante, el cuál solo tiene complejidad $O(n)$. Esto se debe a que dicho flujo se va a ejecutar a lo sumo tantas veces como elementos haya en la cola *habitacionesLimites*. Para saber cuántos elementos hay a lo sumo en esta cola debemos remitirnos a la función *recorrerPorBFS* y ver que aquí sólo se alojaran aquellas habitaciones en las cuales haya una puerta de la que no se tenga la llave correspondiente. Por esto podemos ver que claramente en esta cola habrá a lo sumo n habitaciones.
Sólo basta una aclaración para que la complejidad de este algoritmo quede completa y ésta hace referencia al flujo *while* que engloba a los flujos antes descritos. En un primer acercamiento, uno podría decir que este ciclo se ejecuta a lo sumo n veces, arrojando una complejidad de $O(n)$, haciendo que la complejidad total sea $O(n^3)$ si recordamos que este engloba a una llamada a *recorrerPorBFS* cuya complejidad había resultado ser $O(n^2)$. Pero en realidad, si se mira más detalladamente se puede observar que este flujo va a ejecutarse tantas veces como haga falta para que *recorrerPorBFS* pase por todos las habitaciones (o nodos), es decir, si por ejemplo en el primer llamado a la función *recorrerPorBFS* se pasa por todos los nodos, entonces el ciclo en cuestión va a ejecutarse una única vez. Este análisis entonces nos permite ver que en este algoritmo, la complejidad esta dada por la de la función antes mencionada.

3.4. Detalles de implementación

Para compilar el programa sólo hace falta ejecutar el comando make en consola.

Modo de Uso En consola, utilizar el comando: `./3_ej [entrada] [salida]`.
Los parámetros "entrada" y "salida" son opcionales. Si no se los utiliza, el programa toma por default los archivos "Tp2Ej3.in" como entrada y "Tp2Ej3NUESTRO.out" como salida. En el caso que se los utilice, se le deben pasar 2 nombres de archivos de entrada y salida respectivamente. El archivo de entrada debe existir y debe tener un formato válido, es decir, debe tener el formato descrito en el enunciado de este trabajo práctico. El archivo de salida puede existir o no. En caso de existir, este será completamente sobreescrito.

3.5. Resultados

Para poder analizar el comportamiento del algoritmo propuesto como solución al problema dado fue necesario encontrar una forma de generar de manera automática grafos que lo modelen. Esto resulta de una gran dificultad y se torna mucho más complicado si se quieren generar grafos lo modelen y que además cumplan con ciertas características. Por lo tanto, para generar los casos de prueba se recurrió a un método que utilice números aleatorios. Esto no nos permite elegir alguna característica particular para el modelo final, pero facilita mucho la construcción del generador automático.

Luego de realizar un análisis sobre el comportamiento de los algoritmos *resolver* y *recorrerPorBFS* notamos que en general siempre se van a obtener resultados similares. Para ello se realizaron distintas pruebas:

- Se generaron casos en los que la primer habitación estuviese conectada con la última. Esta se realiza debido a que ambos algoritmos dejan de iterar si, entre otras condiciones, ya se visitó la última habitación.
- Para otros casos, se puso la restricción que ninguna habitación podía estar conectada con la última. Esto significa que no había solución posible.
- Por último, se generaron casos completamente azarosos, con la única precaución que la primer y última habitación no estuviesen unidas entre sí (de otra forma, esto sería igual al primer item).

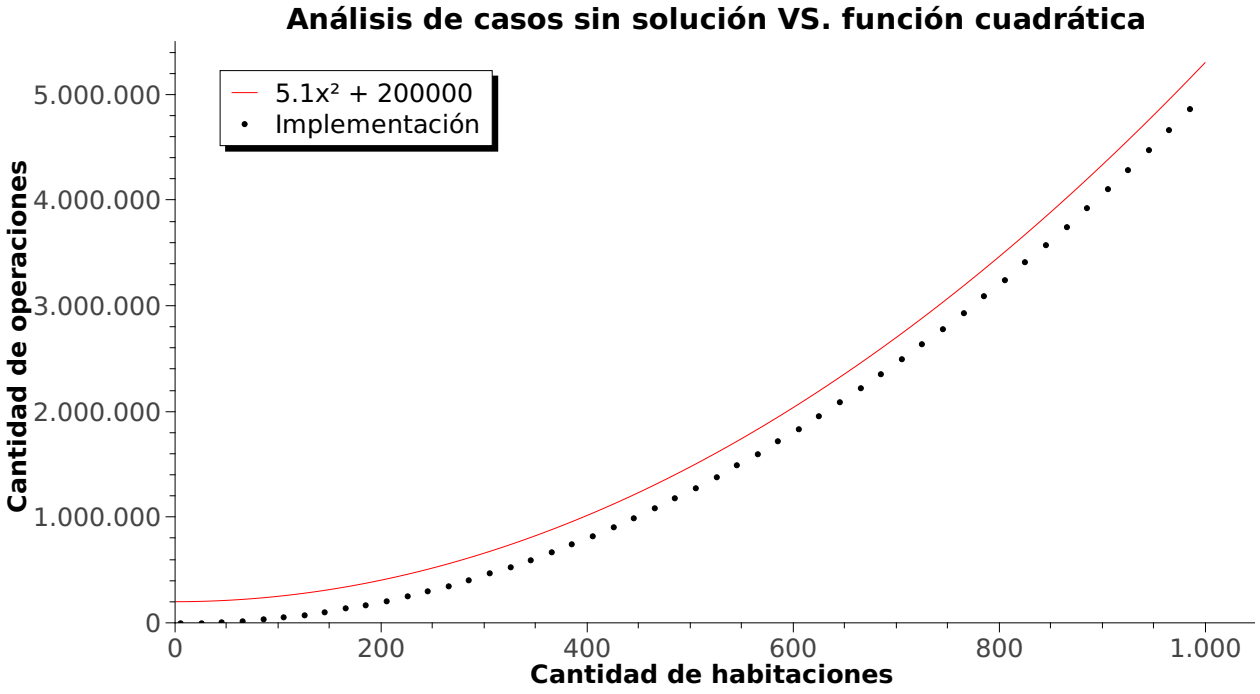
Finalmente, del análisis antes realizado podemos decir que se desprenden algunas hipótesis, las cuales enunciaremos a continuación:

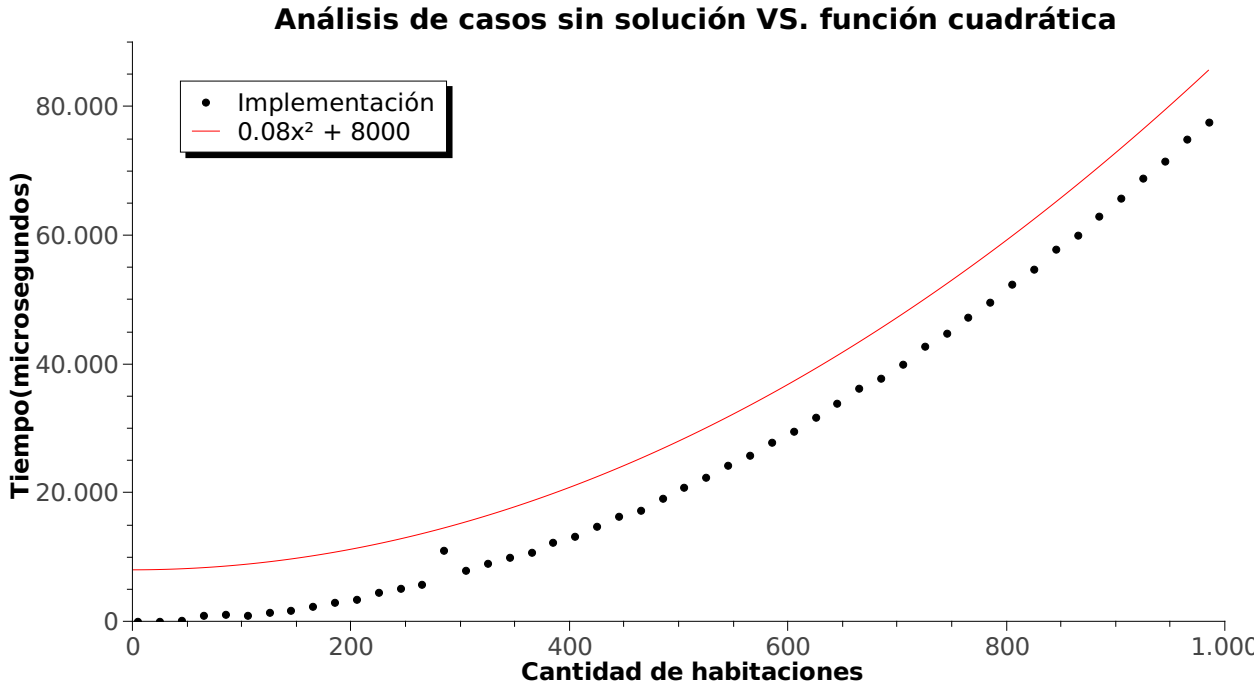
1. Para los casos en los cuales no exista una comunicación directa entre la habitación 1 y la n-ésima, el algoritmo va a presentar una complejidad cuadrática.
2. En los casos en que exista un pasillo que comunique directamente la primer habitación con la última, el algoritmo debería arrojar una complejidad lineal.
3. En aquellos casos azarosos, la complejidad debería ser cuadrática pero en promedio será posible acotarla por una función lineal.

A continuación presentamos los gráficos provenientes de las pruebas realizadas:

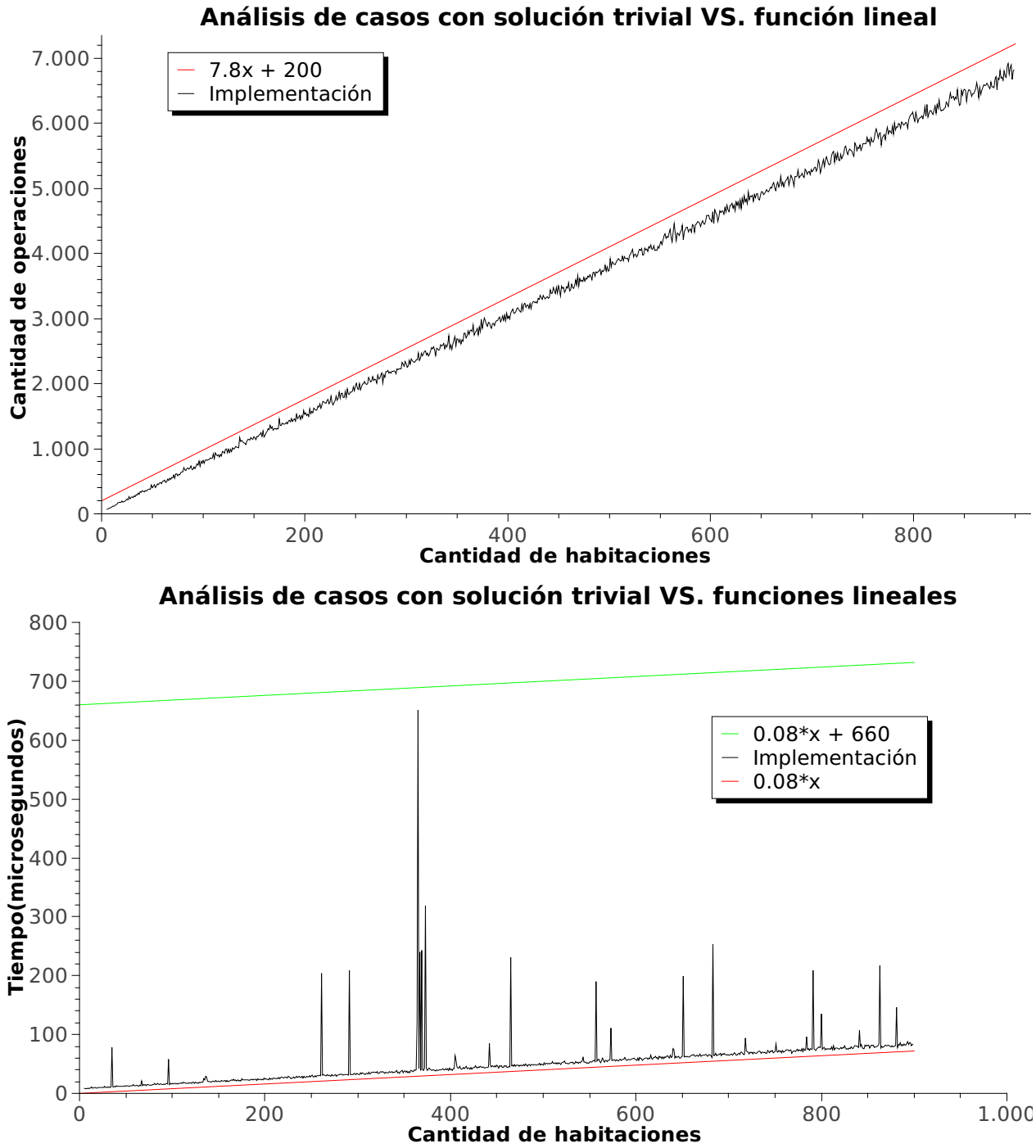
3.6. Debate

3.7. Conclusiones

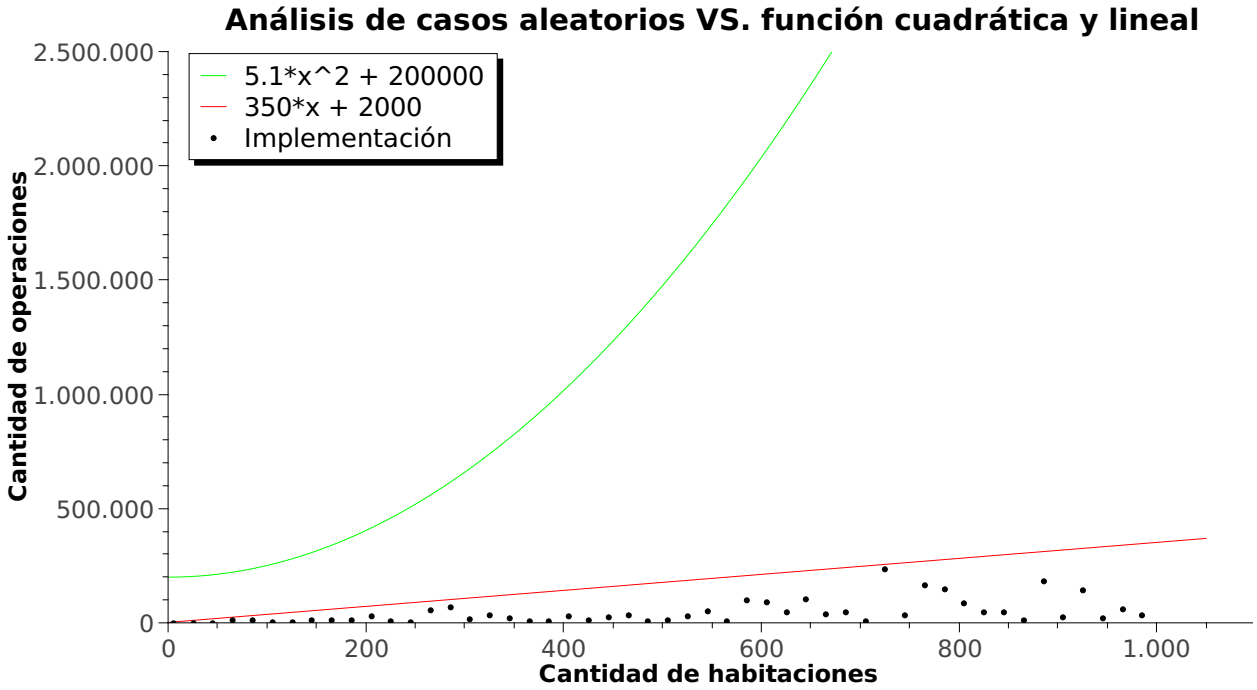


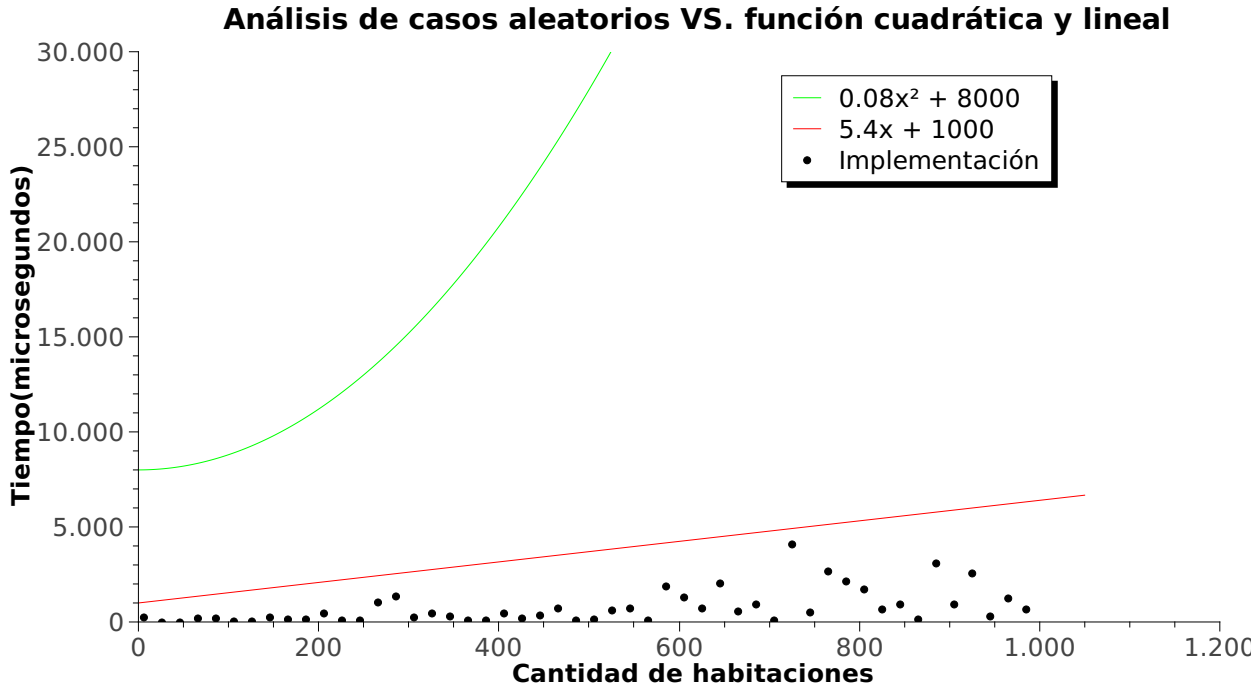


Cuadro 4: Muestran el comportamiento de la cantidad de habitaciones contra la cantidad de operaciones y contra el tiempo respectivamente. Las entradas fueron creadas de modo que no exista un resultado posible, es decir, que ninguna habitación este conectada con la última.



Cuadro 5: Muestran el comportamiento de la cantidad de habitaciones contra la cantidad de operaciones y contra el tiempo respectivamente. Las entradas fueron creadas de modo que haya un resultado trivial, es decir, que la primer habitación este conectada con la última.





Cuadro 6: Muestran el comportamiento de la cantidad de habitaciones contra la cantidad de operaciones y contra el tiempo respectivamente. Las entradas fueron creadas al azar pero sin que exista un resultado trivial.